f responsibilities. Each design pattern focuses on a particular problem or issue. It describes when it applies, whether it can be er design constraints, and the consequences and trade-offs of its entually implement our designs, a design pattern also provides etimes) Smalltalk code to illustrate an implementation.

erns describe object-oriented designs, they are based on practical een implemented in mainstream object-oriented programming alk and C++ rather than procedural languages (Pascal, C, Ada) or -oriented languages (CLOS, Dylan, Self). We chose Smalltalk and easons: Our day-to-day experience has been in these languages, ngly popular.

mming language is important because it influences one's point s assume Smalltalk/C++-level language features, and that choice n and cannot be implemented easily. If we assumed procedural t have included design patterns called "Inheritance," "Encapsu-orphism." Similarly, some of our patterns are supported directly object-oriented languages. CLOS has multi-methods, for example, d for a pattern such as Visitor (page 331). In fact, there are enough Smalltalk and C++ to mean that some patterns can be expressed anguage than the other. (See Iterator (257) for an example.)

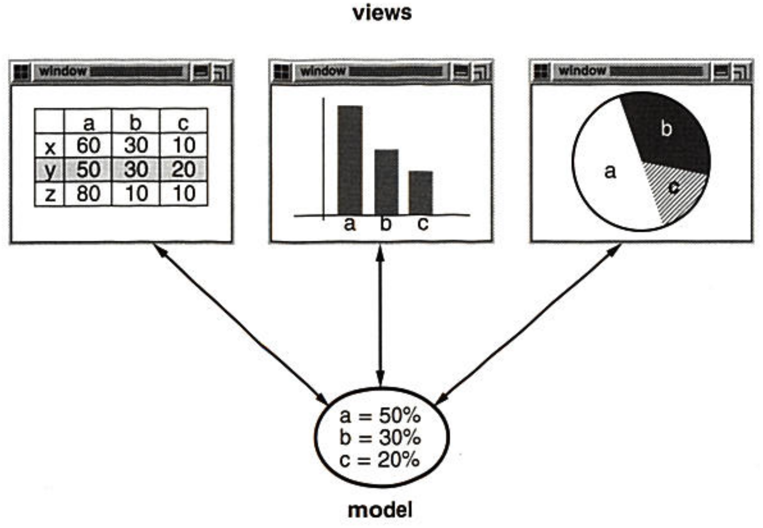
Patterns in Smalltalk MVC

Controller (MVC) triad of classes [KP88] is used to build user inter-0. Looking at the design patterns inside MVC should help you see he term "pattern."

ree kinds of objects. The Model is the application object, the View is tion, and the Controller defines the way the user interface reacts to MVC, user interface designs tended to lump these objects together. em to increase flexibility and reuse.

ews and models by establishing a subscribe/notify protocol between t ensure that its appearance reflects the state of the model. Whenever hanges, the model notifies views that depend on it. In response, each rtunity to update itself. This approach lets you attach multiple views ide different presentations. You can also create new views for a model it.

gram shows a model and three views. (We've left out the controllers e model contains some data values, and the views defining a spread-and pie chart display these data in various ways. The model commu-its values change, and the views communicate with the



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the Observer (page 293) design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite (163) design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

N

of responsibilities. Each design pattern focuses on a particular
problem or issue. It describes when it applies, whether it can be
er design constraints, and the consequences and trade-offs of its
entually implement our designs, a design pattern also provides
etimes) Smalltalk code to illustrate an implementation.

erns describe object-oriented designs, they are based on practical
een implemented in mainstream object-oriented programming
alk and C++ rather than procedural languages (Pascal, C, Ada) or
-oriented languages (CLOS, Dylan, Self). We chose Smalltalk and
easons: Our day-to-day experience has been in these languages,
ngly popular.

mming language is important because it influences one's point
s assume Smalltalk/C++-level language features, and that choice
n and cannot be implemented easily. If we assumed procedural
t have included design patterns called "Inheritance," "Encapsu-
orphism." Similarly, some of our patterns are supported directly
object-oriented languages. CLOS has multi-methods, for example,
d for a pattern such as Visitor (page 331). In fact, there are enough
Smalltalk and C++ to mean that some patterns can be expressed
nguage than the other. (See Iterator (257) for an example.)

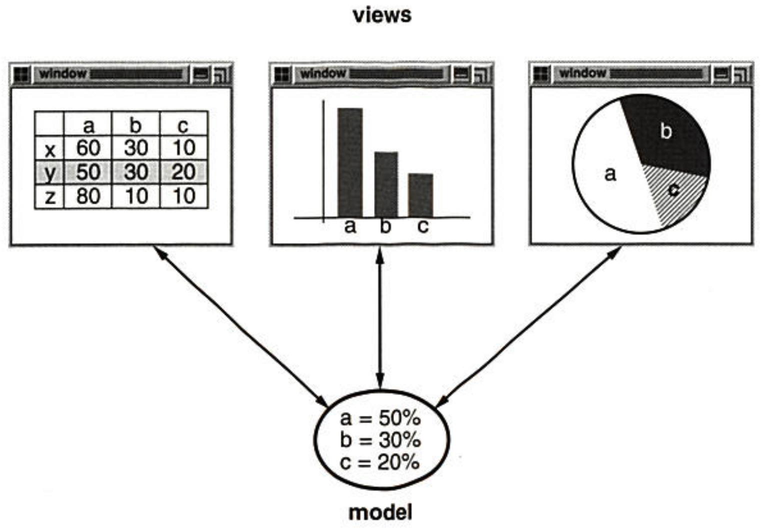
Patterns in Smalltalk MVC

Controller (MVC) triad of classes [KP88] is used to build user inter-
0. Looking at the design patterns inside MVC should help you see
he term "pattern."

ree kinds of objects. The Model is the application object, the View is
tion, and the Controller defines the way the user interface reacts to
MVC, user interface designs tended to lump these objects together.
em to increase flexibility and reuse.

ews and models by establishing a subscribe/notify protocol between
t ensure that its appearance reflects the state of the model. Whenever
hanges, the model notifies views that depend on it. In response, each
rtunity to update itself. This approach lets you attach multiple views
ide different presentations. You can also create new views for a model
it.

gram shows a model and three views. (We've left out the controllers
e model contains some data values, and the views defining a spread-
and pie chart display these data in various ways. The model commu-
values change, and the views communicate with the



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the Observer (page 293) design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite (163) design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.